#11

PATENT

IBM Docket No. **POU920000126US1** **09/619,051**

Certificate of Mailing/Facsimile 37 CFR §1.8(a)
I hereby certify that this correspondence is being:

__X__ deposited with the United States Postal Service as first class mail in an envelope with sufficient postage addressed to the:

_____ facsimile transmitted to **[Fax Number]** to the:

Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on February 25, 2004.

Susan L. Phelps

Susan Phelps
Signature

In the United States Patent and Trademark Office

Date:	February 25, 2004		
In re Application of:	Blackmore et al.	***Filed:***	07/18/2000
For:	**Mechanisms for Efficient Message Passing with Copy Avoidance in a Distributed System Using Advanced Network Devices**		
Serial Number:	09/619,051	***Conf. No.:***	9648
Art Unit:	2153	***Examiner:***	M. Winters

Declaration Under 37 CFR § 1.131

Hon. Commissioner for Patents
P.O. Box 1450, Alexandria, VA 22313-1450

Sir:

I, Robert S. Blackmore, declare the following to the best of my knowledge and recollection:

1. that I am one of the inventors of the invention disclosed and claimed in the above-identified patent application;

2. that I was employed by International Business Machines Corporation in New York at the time of the invention;

3. that the invention described in the above-mentioned application was conceived and reduced to practice in Poughkeepsie, New York, prior to April 24, 2000;

4. that this conception and reduction to practice is evidenced by the attached printout which describes the design for copy avoidance as prescribed in the subject claims;

5. that the feature used to check in the code for copy avoidance in KLAPI was feature #40489;

6. that the fix records for this feature were completed prior to April 24, 2000, and that these records indicated that we had working code which embodied all of the claim steps as set forth in claim 1 of the subject patent application;

7. that I have extracted several of the files checked in under this feature and have verified that they contain the subject copy avoidance code; and

8. that the design for this feature is located in the file /afs/aix/project/design/doc/mohonk/klapi/sysdes.ps which bears a date prior to April 24, 2000.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under

Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Respectfully Submitted,

2-25-2004
Date

ROBERT S. BLACKMORE

MOHONK
Kernel LAPI
System Design

Kevin J. Gildea
Internal Address: LLNA/P963
Poughkeepsie,NY
gildeak at kgnvmc

June 24, 1998: Version 2.3

Abstract

LAPI is a low-level application programming interface that supports one-sided and active message communications. It was initially developed in user space. Kernel LAPI is a version of LAPI that can be used by kernel subsystems. This system design addresses the port of LAPI to the kernel and its exploitation by VSD, HPSN, and GPFS.

Change History and Background

The source files and PostScript version of this document are filed in:

- /afs/aix/project/design/doc/mohonk/klapi/sysdes.*

History of Changes The following versions have been filed:

- **Version 0.1 - 97/10/31 - Initial version.**
- **Version 1.0 - 98/01/15 - Approval version.**

Line Item Tracking Info: The following CMVC feature covers this line item:

- Feature 36411 - GPFS and HPSN Support Using K-LAPI/K-HAL

A Note to Approvers: Approving the design involves entering the "approval" template in the line item tracking feature. See the design process for details, but the template is in: /afs/aix/project/design/process/design.approval.template

You must enter the template in its entirety; automation depends upon it!

A tool is available to make this easy: enter /tools/bin/approve featurenumber and you will be presented with a copy of the template to "vi". It is then added to the feature. (/tools/bin is on the ppsclntxxx machines - or you can mount it from nfsserv.pok.ibm.com:/pps/tools/rs_aix32)

Related Design Documents

- HPSN System Design (K. Gildea)
- HPSN Device Driver - Component Design (May 1997, D. Michailaros)

- GPFS Release 2 - System Design, (1997, R. Curran)
- Colony System Design (P. DiNicola)

Contents

Chapter 1

Requirements

1.1 Requirements as Provided

The overhead of GPFS is dominated by multiple data copies. Elimination of a data copy in GPFS is required to reduce the overhead of GPFS and improve GPFS node throughput.

VSD requires an efficient transport service for communications between clients and servers. Such a transport protocol must support fragmentation and reassembly of large messages, packet flow control, and recovery from lost packets. Message sizes must be at least as large as GPFS block sizes which will be as large as one megabyte in Mohonk. Packet flow control is required to prevent switch congestion that can be caused by many nodes sending to one.

HPSN also requires an efficient transport service with the additional constraint that it can be easily implemented in the firmware of a disk subsystem.

1.2 Requirements as Interpreted

1.2.1 GPFS Copy Avoidance

GPFS maintains a kernel addressable buffer cache to support read ahead and write behind operations. In releases prior to Mohonk, GPFS reads and writes result in two data copies. One copy is between the application buffer, specified on the read or write call, and the GPFS buffer cache. The other copy occurs in VSD-client between the GPFS cache and a CSS IP interface managed cluster buffer. The CSS adapter then uses DMA to access this cluster buffer.

The only data copy in the GPFS path that can be eliminated while preserving the buffer cache is the data copy done by VSD. To eliminate the data copy in VSD, DMA must be used between communication adapters and the GPFS buffer cache.

1.2.2 Transport Service

The benefits of a separate transport service from VSD will only fully be realized if this transport service is available in multiple environments. It first has to be available in the SP switch environment. It must also be available in environments where a fast network is used in place of the SP switch. These requirements can best be met by a transport service that can be mapped directly to the SP switch and use IP as the network layer.

Chapter 2

System Design Overview

Kernel LAPI provides transport services to kernel subsystems that need to communicate via the SP switch. LAPI provides semantics for active messages and one-sided communications. The transport protocol used by LAPI provides message fragmentation and reassembly, packet flow control, and recovery from lost packets. LAPI uses HAL to access the switch packet layer.

Porting LAPI and HAL to the kernel is the subject of this system design along with the exploitation of Kernel LAPI by VSD, HPSN, and GPFS. Support for direct DMA to the GPFS buffer cache is included in the port of LAPI and HAL to the kernel.

2.1 LAPI and HAL Kernel Port

2.1.1 Kernel HAL

Just as in user space, Kernel HAL will provide an abstraction of the SP switch adapter for Kernel LAPI. Most HAL functions port directly to the kernel. Interrupt handling and the open and close functions require the most significant changes. Kernel HAL adds support for super packets (i.e. large contiguous packets that require adapter fragmentation and reassembly functions) and direct DMA.

Most of the HAL open and close functions are implemented in the CSS kernel extension for both user space and kernel. In releases prior to Mohonk, the *open_kclient* function only works with one kernel window for IP. A new version of *open_kclient* that supports IP and and Kernel HAL will be developed. Like IP, the adapter windows for Kernel HAL will use a partition table where logical task is equal to switch node number.

In user space, HAL uses a background thread for interrupt handling. For the kernel version, HAL will register a handler for each open window with the kernel extension. The kernel extension will determine from the adapter window interrupt bit vector which HAL handler to call when it is called as a second level interrupt handler by the CSS device driver. The kernel extension will schedule itself offlevel before calling the registered Kernel HAL handler.

A timer interrupt is also provided by the kernel extension to user space. For Kernel HAL, the kernel extension will also provide this service but it will be based on a kernel watchdog timer.

2.1.2 Kernel LAPI

Kernel LAPI is mostly a recompile of LAPI because all its hardware and most system dependencies are abstracted in HAL. The only exceptions are Kernel LAPI will use kernel locking services in place of the pthread functions it uses in user space and completion handlers will be called directly instead of being scheduled through a separate thread. Kernel LAPI adds support for super packets and direct DMA.

Kernel LAPI needs to support multiple instances for VSD and HPSN to coexist. This feature has not yet been tested in user space.

2.1.3 Direct DMA Support Functions

Background - Current GPFS Pathlengths

Figure 2.1 shows the current flow of an application node in the SP running GPFS using VSD for remote access to disks versus a local access to disk. The figure shows three different buffers involved in a GPFS data access:

ubuf User buffer which is specified by the application making the access.

mbuf IP interface (SP switch) managed cluster buffer.

cache GPFS buffer cache.

[1] In this design there are two copies possible for every file read/write access. All read/write accesses result in data being copied between the GPFS buffer cache and user buffers.

[1] This section only refers to VSD client but its intent is to apply to both VSD client and HPSN device driver.

When an access does not hit in the GPFS buffer cache, GPFS uses VSD to read blocks from remote disks into the buffer cache. GPFS also uses VSD to write back modified blocks from the buffer cache to the remote disks. VSD uses the SP switch via IP protocols to function ship these requests to VSD server nodes.

VSD shares buffers with the SP switch device drivers to totally avoid copies on the VSD server and minimize copies on the VSD client for GPFS. This design is an extension of a general approach in AIX, called interface managed cluster buffers because the IP network interface provides the buffer allocation, for sharing buffers between the socket layer and network interface layer to avoid copies in the IP stack. The SP switch IP interface provides buffers to VSD that are permanently *d_master'd.*

GPFS does not require VSD and can exploit local (shared) disk subsystems. Local disk device drivers in AIX usually avoid copies by using DMA. These device drivers pin and *d_master* the calling subsystems buffers on every read/write call. Thus, when GPFS uses local disks copies are avoided.

To avoid the copy that VSD client does between the GPFS buffer cache and SP switch IP interface buffers, an approach other than using interface managed cluster buffers has to be considered. An approach more like that used by local disks is required, except, *d_master* overheads should be avoided. Conceptually, GPFS needs to pass, through VSD client, pointers and DMA address information for its buffers to the SP switch adapter where DMA can be used to directly access GPFS buffers.

Use of driver managed buffers for VSD server may be preserved, since this design already avoids copies and *d_master* overheads at the server.

Kernel LAPI Direct DMA Support

The objective of using Kernel LAPI with GPFS is to achieve a data flow very similar to the flow shown in Figure 2.1 where data is DMA'd between the SP switch and the GPFS buffer cache. LAPI architecture [3] which includes kernel enhancements supports the concept of subsystem managed buffers, where the subsystem using Kernel LAPI is given the responsibility for allocating communication buffers. Functions are provided by Kernel LAPI to prepare subsystem buffers for DMA, so that when data is transmitted or received, the switch adapter can DMA directly to or from subsystem buffers.

For direct DMA on TBS adapters, the *ptag* argúment must be specified and the receiver must have pre-posted a buffer with the specified ptag value. On Colony adapters, because there is a reassembly area on the adapter, the

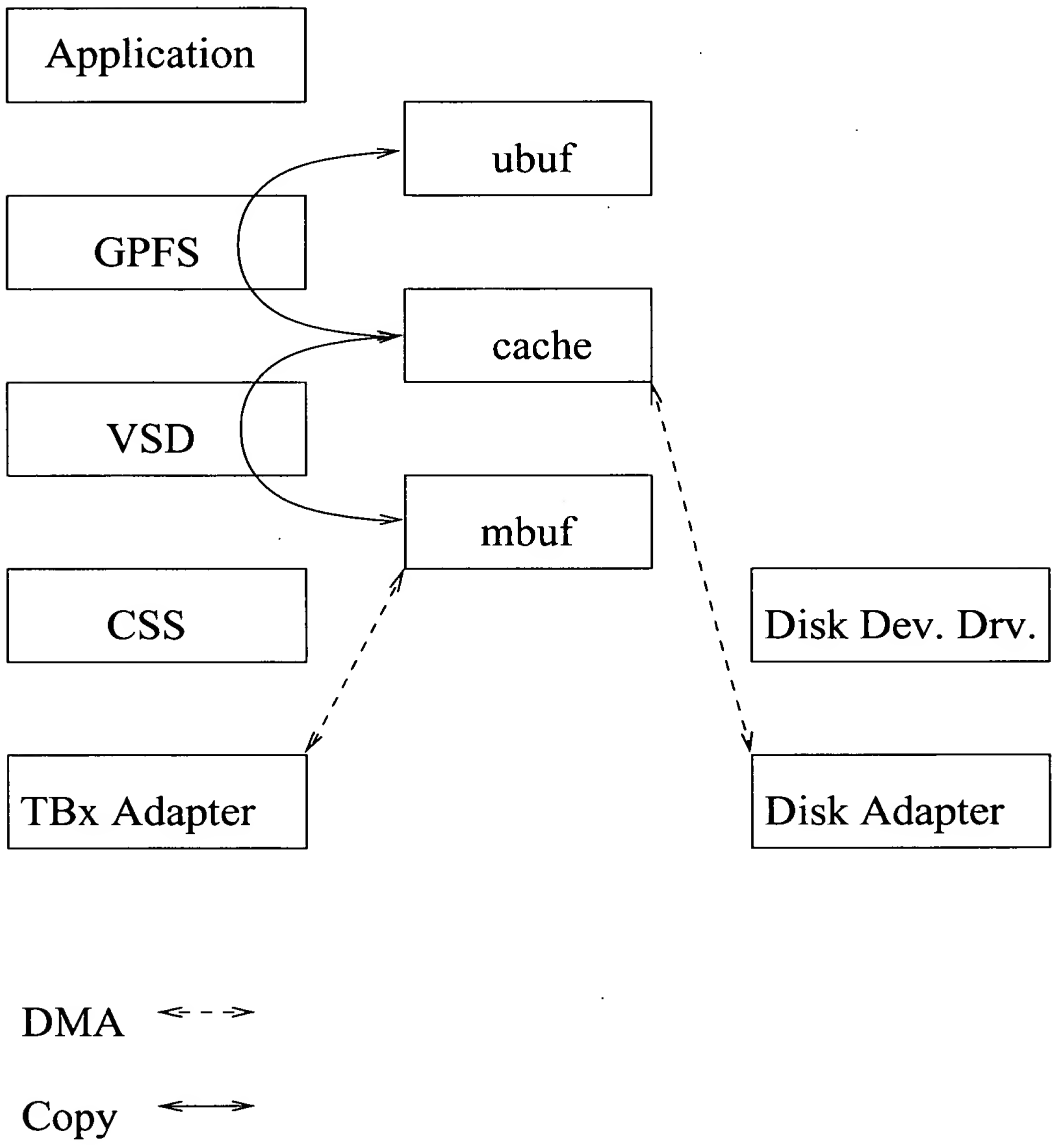

Figure 2.1: GPFS Components

ptag argument does not have to be specified, in which case, data is held in the reassembly area until the subsystems header handler specifies a DMA address and Kernel LAPI initiates the DMA operation.

On TBS adapters, direct DMA functions use TB4 packets and use the same adapter fragmentation and reassembly functions as IP support. For IP, TBS microcode only supports interface managed cluster buffers. A new TB4 packet type is defined to distinguish Kernel LAPI from IP and to indicate the use of the *ptag* argument.

The HPSN device driver and VSD client must provide to GPFS either functions to map its buffers for DMA or map buffers for GPFS. In either case, this must be done within the constraints of available DMA address space.

Kernel HAL Direct DMA Packet Support

Use of ptag on TBS adapters requires the following changes to TB4 packet support. A new TB4 packet type will be introduced to distinquish Kernel HAL packets. This packet type will indicate to adapter microcode that the TB4 message identifier is to be used to store the ptag. On outgoing packets, the ptag value passed to Kernel HAL will be inserted in the TB4 packet header by adapter microcode.

When Kernel HAL type TB4 packets are received, adapter microcode must search a buffer posting table for a buffer with a matching ptag. To post a buffer, system software sends a buffer descriptor to adapter microcode through the send FIFO. When Kernel HAL super packet is reassembled, adapter microcode will return the posted buffer descriptor to sytem software through the receive FIFO.

2.2 VSD Exploitation of Kernel LAPI

In releases prior to Mohonk, VSD uses IP and provides its own transport services. It also shares buffers with the SP switch IP interface. To exploit Kernel LAPI in Mohonk, VSD will utilize Kernel LAPI transport services and provide its own buffer management on the server side. For GPFS, VSD will manage direct DMA to the GPFS buffer cache using Kernel LAPI services.

To utilize Kernel LAPI transport services, VSD will replace calls to its own send and receive functions with calls to the Kernel LAPI active message interface. This includes using *lapi_amsend* and providing header handlers and completion handlers for receiving data.

Kernel LAPI provides message fragmentation and reassembly, packet flow control, and recovery from packets lost in the network as part of its transport service. VSD must provide message flow control to balance the use of server resources and manage recovery from down networks and down nodes. Kernel LAPI provides services to aid in the recovery from down networks and down nodes.

VSD will also provide its own buffer management on the server side. This requires VSD to allocate its own set of pinned kernel buffers and use Kernel LAPI services to prepare these buffer for direct DMA.

2.3 HPSN SCSI Protocol Mapping

The HPSN SCSI device driver and HPSN subsystems communicate using an adaptation of SCSI protocols to TBS. This adaptation is based on the Fibre Channel Protocol for SCSI as outlined in the Seascape Attachment Specification [1]. Four protocol layers are defined: the SCSI application layer based on SCSI-3, the Fibre Channel Protocol (FCP) layer, the transport layer based on LAPI, and the TBS link layer.

The SCSI application layer processing is provided by existing AIX SCSI disk device driver support. The SCSI disk device driver in AIX is an abstraction layer that depends on SCSI adapter device drivers to provide hardware specific communications with SCSI devices.

FCP mapping of SCSI is provided by the HPSN device driver. The HPSN device driver communicates FCP packets to and from Seascape via calls to the Kernel LAPI transport layers.

Kernel LAPI provides the transport layer functions used to send and receive SCSI messages via the SP switch. Kernel LAPI uses the link layer provided by CSS microcode, including, TB4 "super packet" support.

2.4 Functional Flow

2.4.1 Configuration

1. CSS Device Driver Configuration Method begins.
2. Device Driver is loaded and configured.

3. Kernel HAL is loaded and configured.
4. Kernel LAPI is loaded and configured.
5. CSS Device Driver Configuration Method completes.
6. rc.switch begins.
7. Adapter microcode is loaded.
8. rc.switch completes.
9. VSD configuration begins.
10. Kernel LAPI init function is called.
11. Kernel HAL open is called.
12. VSD configuration completes.

2.4.2 Disk Read/Write Scenario

The handling of SCSI commands by the HPSN device driver, Kernel LAPI, CSS microcode, and Seascape are covered in great detail in the Seascape Attachment Specification.

VSD Read/Write Flow

Figure 2.2 and Figure 2.3 show the flow of data for VSD read and VSD write between a GPFS/VSD client and a VSD server using Kernel LAPI. For the write case two alternative flows are shown: one for TBS using rendez-vous and one for Colony using cluster buffers in RAMBUS.

2.4.3 Direct DMA Packet Handling

Special handling is required in Kernel HAL to receive packets for direct DMA. On TBS adapters, adapter microcode uses the ptag field in the packet headers along with other TB4 header fields to DMA data directly into a subsystem managed buffer. On Colony, microcode directs packet reassebly into RAMBUS while system software initiates DMA or copy from RAMBUS into subsystem managed buffers.

Figure 2.4 shows the flow of Kernel HAL packets in adapter microcode. Figure 2.5 shows the flow of Kernel HAL packets on the system side of the

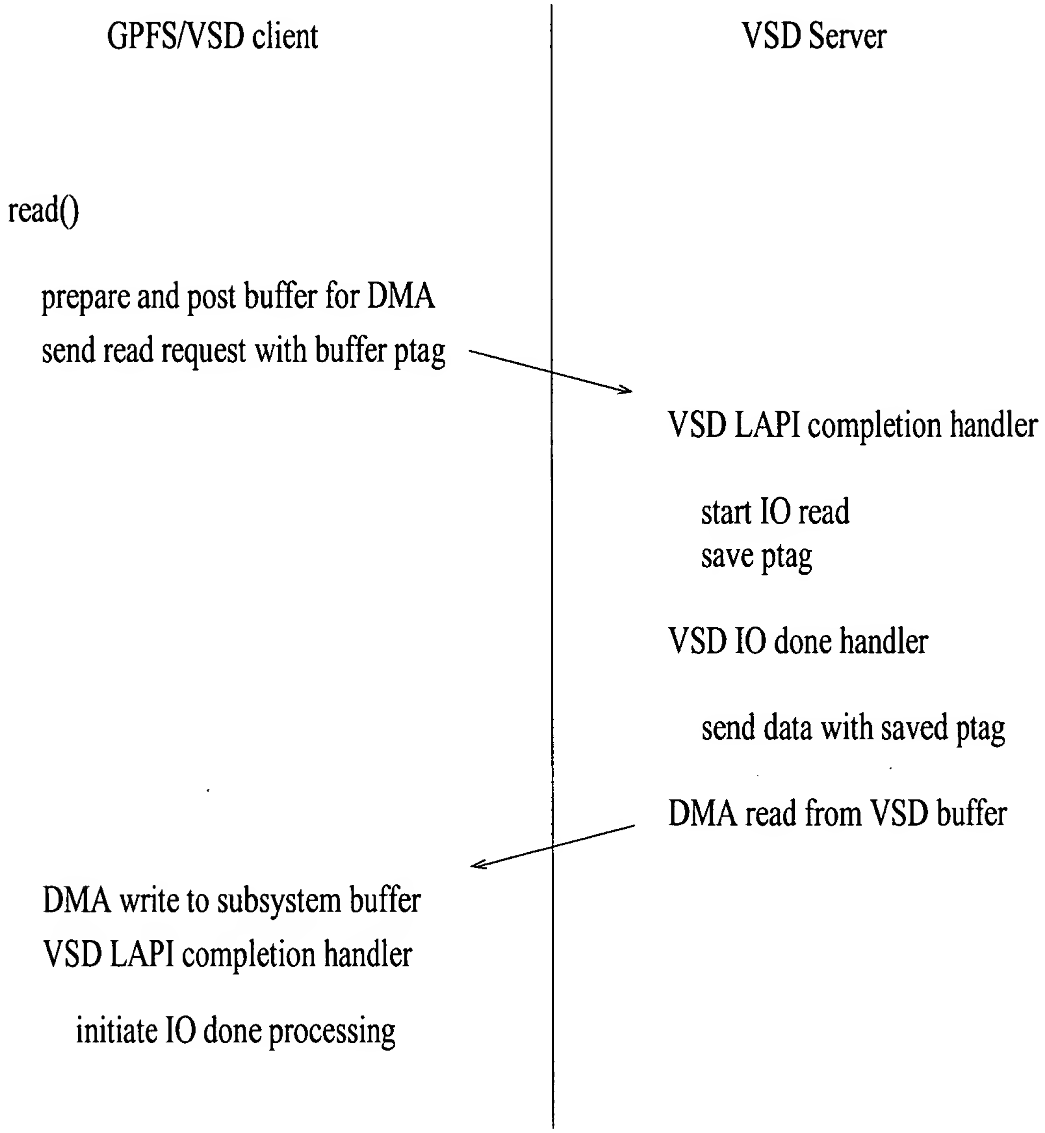
GPFS/VSD client
VSD Server
read()
prepare and post buffer for DMA
send read request with buffer ptag
VSD LAPI completion handler
start IO read
save ptag
VSD IO done handler
send data with saved ptag
DMA read from VSD buffer
DMA write to subsystem buffer
VSD LAPI completion handler
initiate IO done processing

Figure 2.2: VSD Read Flow

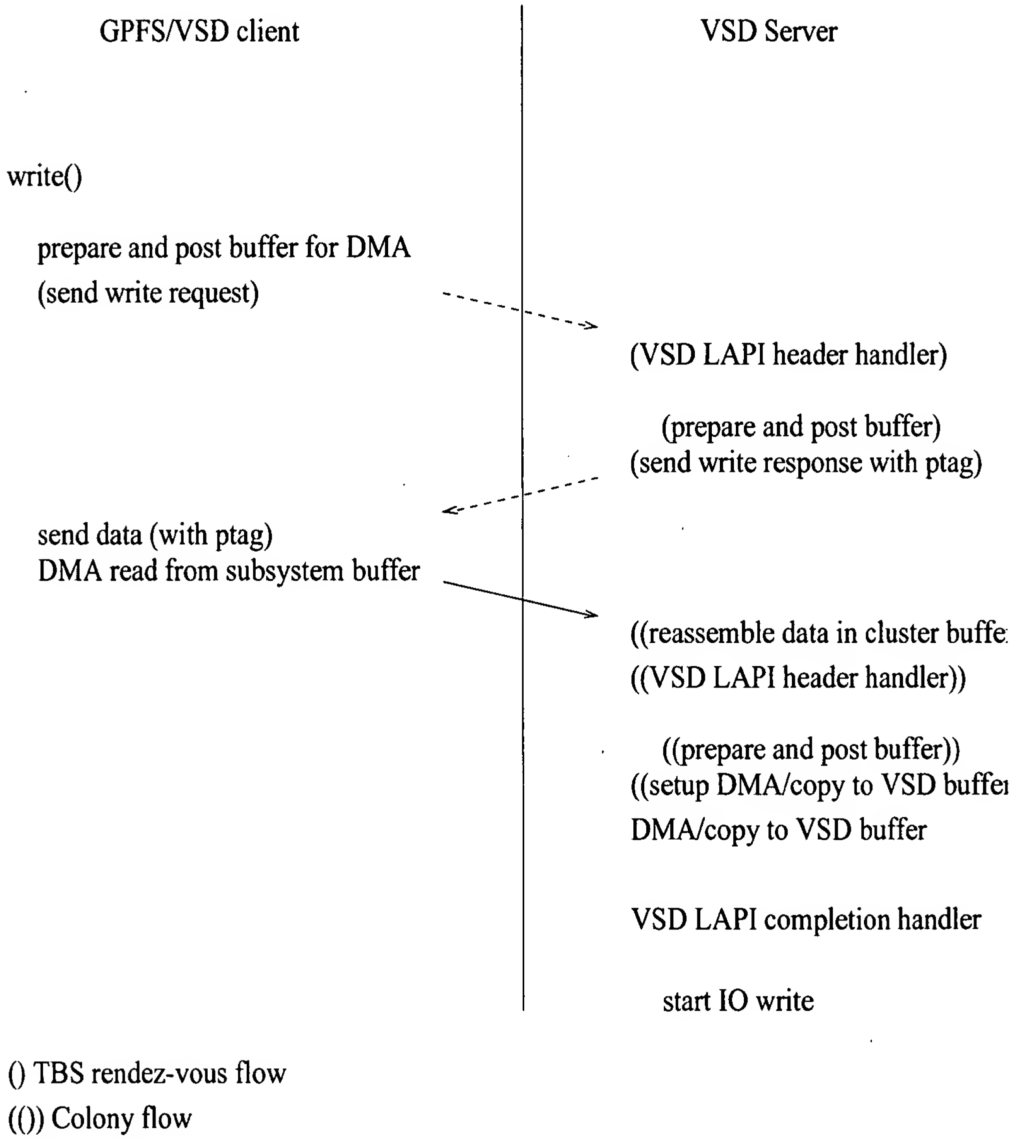
GPFS/VSD client
VSD Server
write()
prepare and post buffer for DMA
(send write request)
(VSD LAPI header handler)
(prepare and post buffer)
(send write response with ptag)
send data (with ptag)
DMA read from subsystem buffer
((reassemble data in cluster buffe
((VSD LAPI header handler))
((prepare and post buffer))
((setup DMA/copy to VSD buffei
DMA/copy to VSD buffer
VSD LAPI completion handler
start IO write
() TBS rendez-vous flow
(()) Colony flow

Figure 2.3: VSD Write Flow

adapter interface. In both flows, existing flows are distinquished from new flows required for direct DMA on TBS adapters and the Colony adapter.

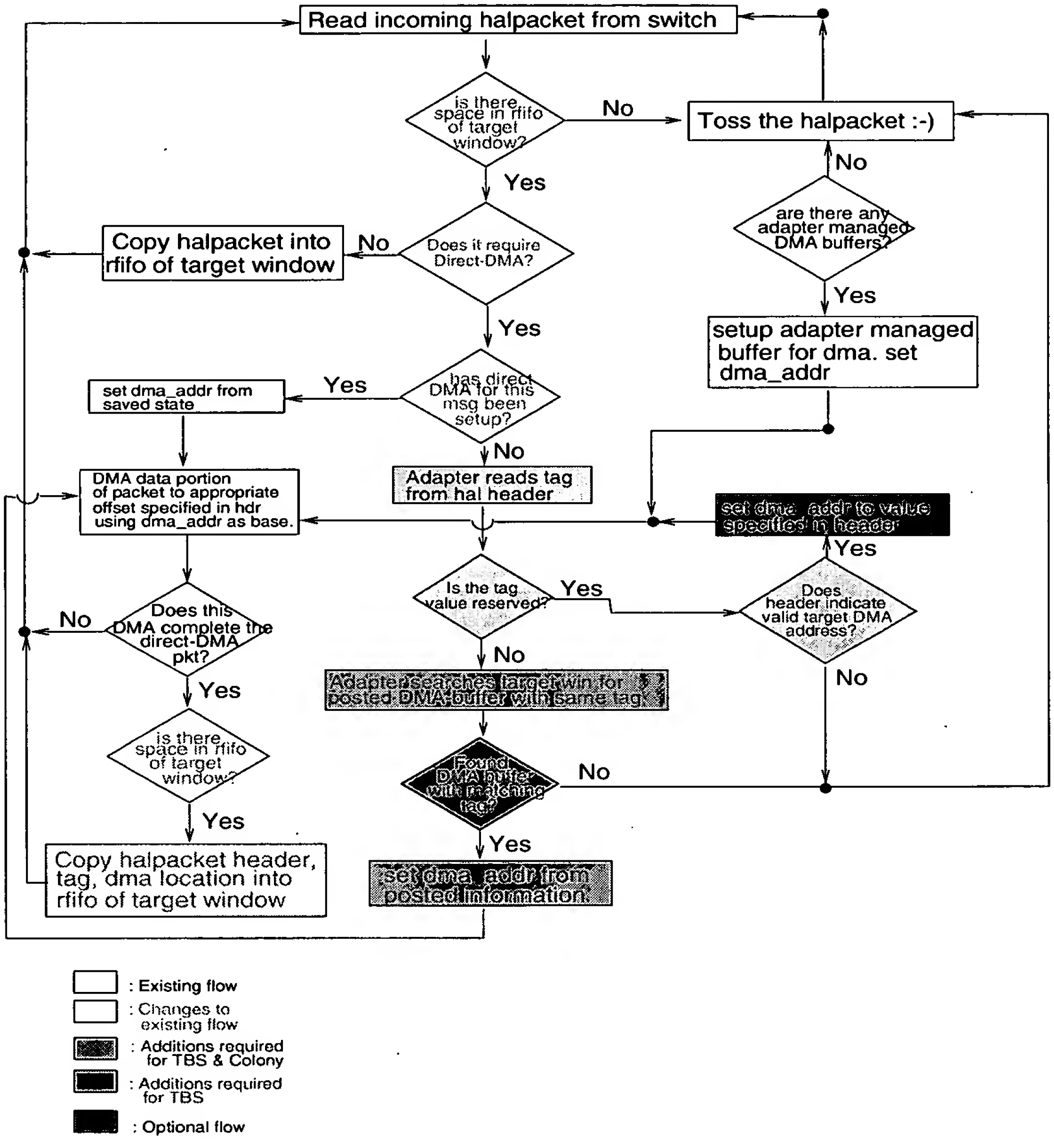
Read incoming halpacket from switch
is there space in rfifo of target window?
No
Toss the halpacket :-)
Yes
No
are there any adapter managed DMA buffers?
Copy halpacket into rfifo of target window
No
Does it require Direct-DMA?
Yes
Yes
setup adapter managed buffer for dma. set dma_addr
set dma_addr from saved state
Yes
has direct DMA for this msg been setup?
No
DMA data portion of packet to appropriate offset specified in hdr using dma_addr as base.
Adapter reads tag from hal header
set dma_addr to value specified in header
Yes
No
Does this DMA complete the direct-DMA pkt?
Is the tag value reserved?
Yes
Does header indicate valid target DMA address?
No
Yes
Adapter searches target win for posted DMA buffer with same tag
No
is there space in rfifo of target window?
Found DMA buffer with matching tag?
No
Yes
Yes
Copy halpacket header, tag, dma location into rfifo of target window
set dma_addr from posted information.
: Existing flow
: Changes to existing flow
: Additions required for TBS & Colony
: Additions required for TBS
: Optional flow

Figure 2.4: Kernel HAL Packet Flow - Microcode

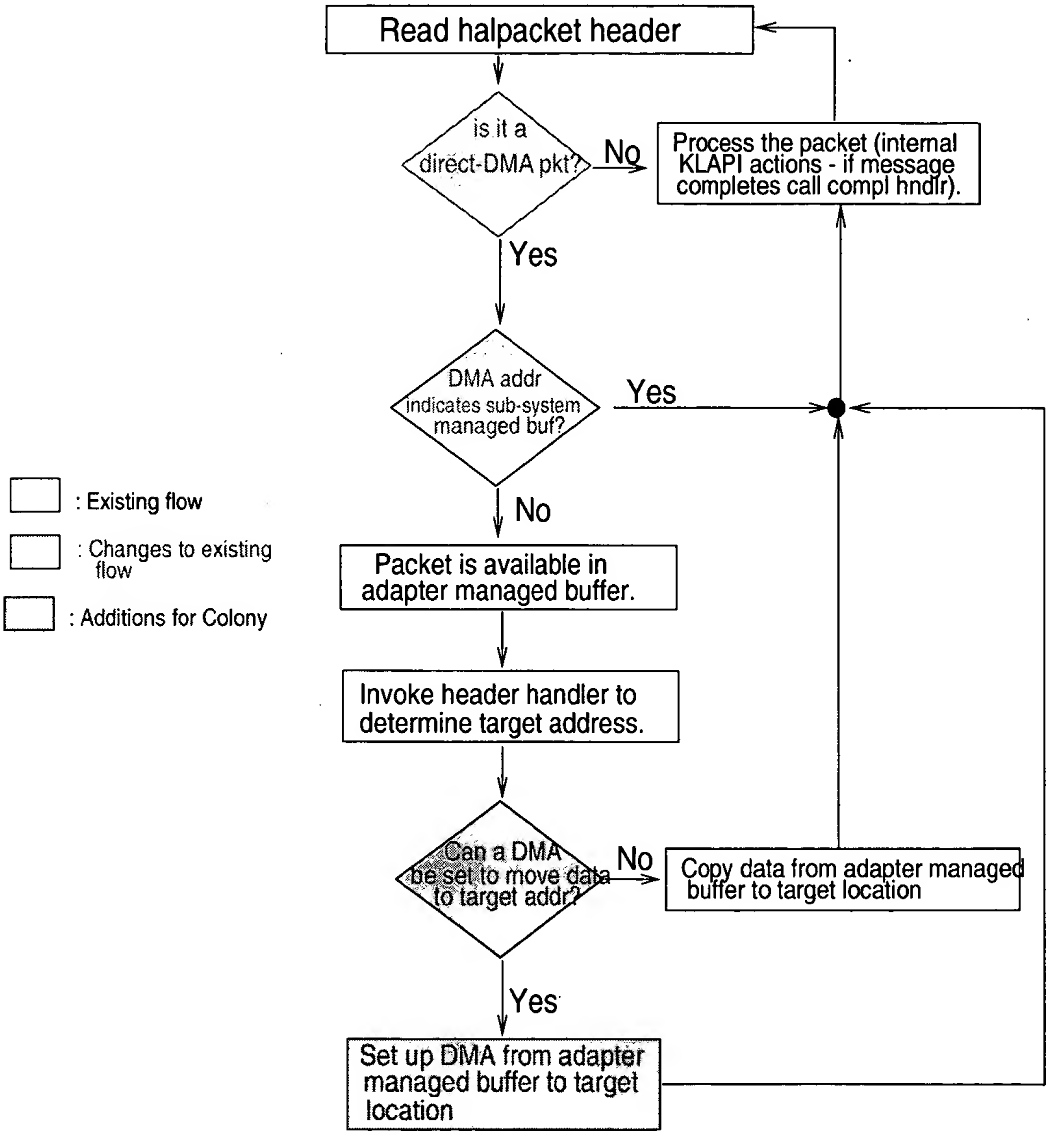
Read halpacket header
is it a direct-DMA pkt?
No
Process the packet (internal KLAPI actions - if message completes call compl hndlr).
Yes
DMA addr indicates sub-system managed buf?
Yes
No
: Existing flow
: Changes to existing flow
: Additions for Colony
Packet is available in adapter managed buffer.
Invoke header handler to determine target address.
Can a DMA be set to move data to target addr?
No
Copy data from adapter managed buffer to target location
Yes
Set up DMA from adapter managed buffer to target location

Figure 2.5: Kernel HAL Packet Flow - System

Chapter 3

High Level Direction for Components

3.1 Dependent Node Architecture

3.1.1 Kernel HAL Packet Type

- Add a new TB4 packet type for Kernel HAL to distinguish from IP packets. Kernel HAL packets are not routable outside the SP switch.

3.2 CSS

3.2.1 Kernel Extension

- Add support for multiple kernel windows to *open_kclient* and *close_kclient*. A window select parameter should be added to *open_kclient*. IP will continue to be fixed to window 0 and service to window 1. Windows 2-n will be additional windows reserved for Kernel HAL. The number of windows reserved for Kernel HAL will be determined by component design but is required to be at least one. The number of user space windows will not be affected by this system design. [1]

- *d_master* memory provided by Kernel HAL and set up for use as send and receive FIFO's.

[1] It is advised that in Colony the allocation of windows become less static.

- For kernel windows, *open_kclient* also DMA masters memory for use as cluster buffers (i.e. sendpool and recvpool.) Kernel HAL windows will also be allocated the same amount of DMA space by *open_kclient* but the *d_master* operation should be skipped.
- Make *open_kclient* parameters consistent with *open_uclient* wherever possible.
- Provide demultiplexing of adapter interrupts to Kernel HAL windows, calling a registered handler for each window off level.
- Provide the timer service to Kernel HAL based on watchdog timer.
- Provide a switch fault signal to Kernel HAL.
- Apply fix to defect 29071 to kernel windows.
- An error should be returned by *open_kclient* if adapter microcode has not been loaded.

3.2.2 CSS TB3 and TB3MX Microcode

- Enable super packet support for windows other than WIN_IP.
- Enhance receive buffer management so a Kernel LAPI window can keep a list of posted buffers by *ptag*.
- For Kernel LAPI type TB4 packets, match *ptag* field (i.e. TB4 message identifier) to a posted buffer.
- Add support to post ptag buffer descriptors and return buffer descriptors when super packets are complete.

3.2.3 CSS Device Driver

- Load and configure Kernel HAL and Kernel LAPI from the configuration method after the device driver is configured.
- Provide additional DMA address space directly to Kernel HAL for use in support of Direct DMA.

3.3 Kernel LAPI and HAL

3.3.1 Kernel HAL

- Build as separate kernel extension.
- Maintain a single HAL code base for both kernel and user spaces with _KERNEL compile option defined for kernel mode.
- Use offlevel interrupt provided by CSS Kernel Extension in place of interrupt thread.
- Use kernel locks in place of Pthread mutex locks.
- Provide locking to serialize interrupt handling.
- Use BUSMEM_ATT and BUSMEM_DET for PIO access to adapter. An exception handler must also be used for PIO's.
- Allocate from the kernel pinned heap memory to be used for send and receive FIFO's.
- Add direct DMA functions outlined by architecture [3][4].
- Kernel HAL functions will be exported to other kernel subsystems but not as system calls.

3.3.2 Kernel LAPI

- Build as separate kernel extension.
- Maintain a single LAPI code base for both kernel and user spaces with _KERNEL compile option defined for kernel mode.
- Call completion handler from offlevel interrupt instead of scheduling a separate completion handler thread.
- Use kernel locks in place of Pthread mutex locks.
- Add direct DMA functions outlined by architecture [3][4].
- Kernel LAPI functions will be exported to other kernel subsystems but not as system calls.
- Kernel LAPI will be instantiated with 512 tasks, one task per node.
- Define and use ptag buffer descriptor format.

3.4 VSD

- Port VSD functionality to Kernel LAPI.
- Create a Kernel LAPI instance for VSD communications.
- Provide buffer management services on the server side, including, allocating pinned buffers and setup of these buffers for DMA using Kernel LAPI services.
- Provide direct DMA into GPFS buffer cache using Kernel LAPI services.
- Provide message level flow control to balance server resources.
- Provide recovery from network failures and node failures.
- In multiple switch adapter systems (e.g. Colony), create one Kernel LAPI instance for each adapter.

Chapter 4

System Design Template

4.1 Interface Semantics

Kernel LAPI is defined in [3]. Kernel HAL is defined in [4].

4.2 Installation and Packaging

Kernel LAPI and Kernel HAL will be packaged and installed as part of the CSS component of PSSP.

4.3 Rationale

Use of Kernel LAPI by VSD and HPSN ensures communications is efficient, portable, and separated from disk functions by an architected interface. Kernel LAPI provides a hardware independent transport layer that can be mapped directly to the switch adapters for maximal efficiency or to IP for maximal flexibility. It provides necessary functions for reliability, flow control, fragmentation, and reassembly.

The separation of communications functions from the disk functions into a transport layer will facilitate development and testing. It will also create opportunities to reuse these communication functions in other applications.

By focusing initial HPSN support on a version of Kernel LAPI that maps directly to the switch adapter, direct DMA into the calling subsystems buffers will be achieved and the overhead of a network layer will be eliminated. These features will significantly reduce IO overhead in applications using switch attached storage.

4.4 Dependencies

4.4.1 Relationships to Other Designs

Direct DMA support in HAL and LAPI must be factored into the Colony system design. The use of *ptag* and buffer matching will not be supported. Instead, RAMBUS cluster buffers will be used to stage Kernel LAPI messages. When a full message is available in RAMBUS, Kernel LAPI will call a subsystem (VSD) defined header handler to specify the final destination of the data. Ideally, DMA will be used to move data from RAMBUS to a subsystem assigned buffer.

4.5 Scaling Considerations

4.5.1 Current Design Point

This design does not increase the number of nodes allowed in the system. Kernel LAPI being based on LAPI, is subject to the same scaling factors as LAPI.

4.5.2 MP Enablement

Kernel LAPI will be MP-safe and exploiting.

When direct DMA is used, Kernel LAPI lock regions are minimized and a single instance should sustain the full bandwidth of a single switch adapter. If direct DMA is not used, the Kernel LAPI lock regions will include a data copy, significantly increasing time locks are held and limiting the bandwidth of a single instance to the twice the copy rate of the CPU's (i.e. Kernel LAPI will allow the send and receive paths to run in parallel.)

To fully exploit an MP, multiple LAPI instances can be created by exploiting subsystems.

4.5.3 Large N-Way RPQ Systems

4.6 Alternative Designs

4.7 Supported Hardware

Kernel LAPI and Kernel HAL will be developed initially for TB3 and TB3MX adapters. It will also be supported on Colony. TB2 adapters will not be supported.

4.8 Performance Objectives

Kernel LAPI performance will meet or exceed all performance aspects of a single LAPI user space job.

Using Direct DMA, GPFS bandwidth will improve 25 percent due to the avoidance of a second data copy.

4.9 Standards and Policies

4.9.1 External

4.9.2 AIX

4.9.3 PPS Internal

Policy	Applicable to this Design?	This Design Complies?
Error Logging	Yes	Yes
Reliable Daemons	No	
Commands	No	
Message Guidelines	Yes	Yes
README	No	

4.9.4 NLS

Kernel LAPI will use AIX/6000 NLS and comply with SP Message policy and style guide. The rest of the new support will be consistent with the NLS capabilities of the changed components.

4.10 Migration, Coexistence, and Compatibility

The Kernel LAPI exploiting version of VSD will also continue to support its IP protocol so that nodes running Mohonk VSD can interoperate with nodes running older versions of VSD. This will allow single nodes to be migrated to the newer Mohonk VSD. Kernel LAPI protocols will either only be used between nodes already migrated to Mohonk or only after all nodes have been migrated to Mohonk.

4.10.1 Portability to Other Platforms

4.11 Reliability, Availability, and Serviceability

4.11.1 Error Reporting

Kernel LAPI, and Kernel HAL will comply with SP Error Logging policy.

4.12 Security

Kernel HAL packets are not routable and can only be generated by kernel code.

If Kernel LAPI is also mapped to IP, then IP filtering will be used to provide security.

4.13 Information Development and Usability

4.13.1 Tasks

None yet. Kernel LAPI functions are not exposed to users and the operation of Kernel LAPI is hidden under the VSD subsystem. There may be minor hits if we start to add tunable Kernel LAPI configuration parameters.

4.13.2 Summary of Externals

4.13.3 Usability

4.14 Invention Protection

The LAPI interface is also extended, in a possibly unique way, to support Direct DMA. Kernel LAPI and microcode functions to support Direct DMA should be considered for invention protection.

4.15 Issues

4.15.1 VSD Requirements

The following list of issues where raised during a Kernel LAPI review with the VSD and HPSN team. These issues must be addressed in the system design and architecture. They will be held in this section until the system design and architecture can be synchronized.

- VSD requires an interrupt at the origin when an origin buffer is complete (i.e. when the origin_ctr is incremented.) This is an architecture issue.
- How much resources does each LAPI channel use? Clarification is required.
- Kernel LAPI does not recover from loss of a communication adapter. It is the using subsystems responsibility to keep enough state to recover from a communication adapter failure and (possible) redirect to another LAPI channel using a different communication adapter. This is a clarification.
- VSD requires functions to clean up state associated with a down destination. This is an architecture issue.
- Need to formalize the use of predefined handlers versus address init function. This is an architecture issue.
- Need to examine the way DMA addresses are passed as arguments. Should xmem descriptors be used? This is an architecture issue.
- Need to define a clean way to handle the DMA master operation for GPFS buffers. Ideally, this should be hidden from GPFS. This is a VSD and HPSN component design issue.

Kernel LAPI Support for IP

It would be very beneficial if all VSD communications operations were handled by Kernel LAPI for both switch environments and IP environments. Everyone agreed that if Kernel LAPI is efficient in IP environments then VSD could eliminate its direct path to IP. This needs to be verified.

VSD Direct DMA

VSD wants to use only one method for direct DMA, assign receive buffer location from the header handler. On Colony this operation can be supported without an extra copy. On TBS adapters, to avoid the extra data copy, Kernel LAPI will perform a rendez-vous prior to sending the data, call the VSD header handler during the rendez-vous, post the VSD assigned buffer with a *ptag*, and reply to the rendez-vous with the *ptag* so the data can be sent with the *ptag*. For short messages on TBS an extra data copy is acceptable to avoid the rendez-vous delay. This approach will give VSD a consistent interface for all environments that allows it to assign buffers based on message header information.

4.15.2 Bill Tuel's Preliminary Review Comments

The original note lines are preceded with >> and with the authors responses imbedded.

```
>>Date: 20 November 97, 08:53:23 EST
>>From: TUEL      at KGNVMC
>>To:   GILDEAK, RAMA at MHV, GAUTAM at MHV
>>Re: Questions on Mohonk Kernel LAPI System Design
>>
>>If Kernel LAPI takes user-space windows away from MPI or non-kernel
>>LAPI, then LoadLeveler needs to be told how many user-space windows
>>can be scheduled.  Thus there must be a hit to LoadLeveler which
>>should be mentioned.  Also, the possible interaction of multiple
>>KLAPI instances with the ability to run user-space jobs on the same
>>node should be noted, since it looks like they compete for the
>>same adapter windows.
>>

The system design has been changed, requiring at least one additional
adapter window for KLAPI use.  This eliminates the requirement for
interaction with LoadLeveler in Mohonk when VSD is the only user of
KLAPI.  In future releases, if there are more KLAPI users and
the additional adapter windows cannot be provided without impact
to user space communications, this will have to be considered again.

>>In section 2.1.2 is says that the KLAPI completion handlers will be
>>called directly - is this in an interrupt context or on the user's
>>kernel thread?  Is there a way to ensure that these don't block?
>>

Completion handlers will run at the offlevel interrupt level.  This
```

enables higher priority (hardware) interrupts to run while completion handlers are active. Still the process environment is not enabled. VSD runs are the offlevel interrupt level today. Anyone using completion handlers is obliged to follow AIX recommendations on offlevel interrupt processing.

>>The footnote in section 3.2.1 suggests that the window allocation become
>>less static? Is there anyone working on that design? Is this a likely
>>prospect for Mohonk?
>>

Not sure of the status of this as the Colony system design is still in process.

>>In section 3.3.1 it says that the receive FIFO's are allocated from
>>the pinned kernel heap. Is there enough room there to handle the
>>amount of space required? How much space is required?
>>

The receive FIFO's do not have to be large in the case of KLAPI since most data will be transmitted using direct DMA to subsystem (e.g. VSD) supplied buffers. We are concerned about the amount of memory VSD will need to manage. We have done analysis for VSD in the ASCI system and opened a requirement against AIX (POR 104165) to ensure either enough heap space or move VSD buffers off the kernel heap. For ASCI, VSD buffer use is expected to be about 100 MB.

>>In section 3.3.2 it says that KLAPI will be instantiated with 512 tasks.
>>Should that be 512 nodes, with one kernel task per node? If multiple
>>windows/LAPI instances were provided, what would be the addressing
>>structure? Would it ever allow a communication domain larger than
>>512?
>>

Kernel LAPI will be instantiated with 512 tasks, one task per node. Each Kernel LAPI instance will be instantiated the same. LAPI functions include a handle parameter which is used to address a specific LAPI instance.

```
>>In section 4.5.2 it says that a single LAPI instance should sustain the
>>full bandwidth of a single switch adapter.  I'm under the impression
>>that a single Nighthawk processor can't drive a Colony adapter with
>>two ports and two fabrics.  Does this statement imply that multiple
>>processors will somehow cooperate to drive data faster?
>>

In user space, it is true that a single Nighthawk processor can't drive
a Colony adapter at full bandwidth, at least as long as the user space
path includes a data copy.  The direct DMA approach in Kernel LAPI
eliminates the data copy from the CPU communication path, enabling
higher per processor bandwidths.

>>Bill Tuel
```

4.16 Future Considerations

Bibliography

[1] "Seascape TB3 (SP2) Interface Attachment", IBM Storage Subsystems Division, May 22, 1997, SJEVM5(MNH).

[2] "Attaching Dependent Nodes to the SP Switch", IBM RS/6000 Division, 1997, Jay H. Benjamin.

[3] "LAPI Architecture Definition", IBM RS/6000 Division, 1998, /afs/aix/arch/lapi/lapi.ps

[4] "HAL Architecture Definition", IBM RS/6000 Division, 1998, /afs/aix/arch/lapi/hal.ps

End of Document